Preprint
UCRL-JC-146819

Interactive View-Dependent Rendering of Large Iso-Surfaces

B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci, K.I. Joy

This article was submitted to
SIGGraph 2002 29th International Conference on Computer Graphics and Interactive Techniques, San Antonio, TX, July 21-26, 2002

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

January 11, 2002

Interactive View-Dependent Rendering of Large Iso-Surfaces

Benjamin Gregorski* Mark Duchaineau* Peter Lindstrom*
Valerio Pascucci*
Center for Applied Scienti£c Computing
Lawrence Livermore National Laboratory

Kenneth I. Joy†
Center for Image Processing and Integrated Computing
University of California, Davis

Abstract

We present an algorithm for interactively extracting and rendering iso-surfaces of large volume datasets in a view-dependent optimal fashion. A recursive tetrahedral mesh subdivision scheme, based on longest edge bisection, is used to hierarchically decompose the data into a multi-resolution structure. This data structure allows fast extraction of arbitrary iso-surfaces to within user specified view-dependent error bounds. A compact encoding of the mesh subdivision optimizes memory usage and processor performance necessary for large datasets. A data layout scheme based on hierarchical space filling curves provides optimal access to the data in a cache coherent manner.

Keywords: View-Dependent Rendering, Iso-Surfaces, Multiresolution Tetrahedral Meshes, Multiresolution Techniques

1 Introduction

Iso-surfaces are a fundamental method for visualizing volume datasets. Iso-surface extraction is a well developed field starting with [Lorensen and Cline 1987]. Fast extraction methods have been developed in [Bajaj et al. 1996], [Chiang and Silva 1997], and [Bajaj et al. 1999]. Recent research on distance fields and volumetric representations for objects [Frisken et al. 2000], [Ferley et al. 2000], [Kobbelt et al. 2001] and reconstructing objects using implicit functions [Carr et al. 2001] shows how volumetric representations can be effectively used to describe objects. Given a volume representation of an object such as a CT scan, MRI scan, or distance field, the object can be rendered by extracting iso-contours from the volume. For large volumes of size 512^3 or greater, multiresolution view-dependent techniques that exploit frame to frame coherence are essential for interactive rendering and exploration.

We use the subdivision of a tetrahedral mesh via longest edge bisection to build a multi-resolution hierarchy of the volume dataset.

*{gregorski1,duchaine,pl,pascucci}@llnl.gov
†kijoy@ucdavis.edu

Figure 1: Cyberware Igea Model. Size = 512^3 voxels, Screen Error = 0.5 pixels, 70K triangles

The tet subdivision scheme that we use is described in detail in [Zhou et al. 1997], [Gerstner and Rumpf 2000], and [Gerstner and Pajarola 2000]. We combine the bottom-up and top-down traversal schemes presented in these papers to create an adaptively refinable tetrahedral mesh. This adaptive mesh supports the dual queue split/merge algorithm as described in the ROAM system[Duchaineau et al. 1997] for view-dependent terrain visualization. It has fast coarsening and refinement operations which allow for strict frame to frame triangle counts, progressive optimization of mesh quality, and guaranteed frame rates. All of these are essential for interactive view-dependent rendering.

Each tet in our mesh approximates a portion of the volume using a linear interpolant. Volume representations using tetrahedral meshes have been developed in [Trotts et al. 1999], [Cignoni et al. 1997], and [Cignoni et al. 2000]. At run time, the split/merge refinement algorithm is used to create a lower resolution dataset that approximates the original dataset to within a given error tolerance. The error tolerance is a measure of how much an iso-surface drawn through the lower resolution dataset deviates from the finest

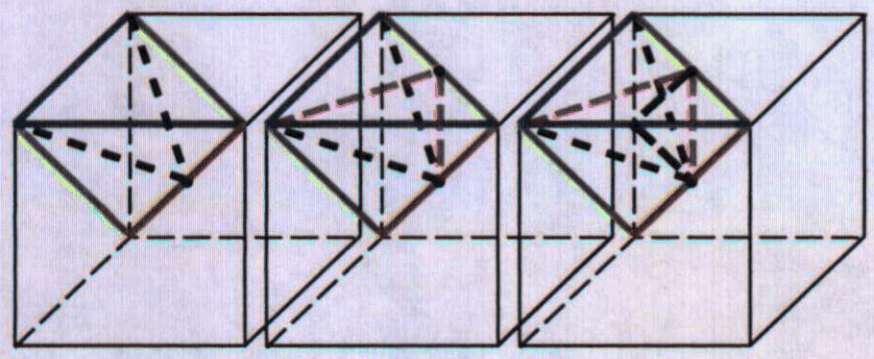

Figure 2: Three Subdivision Phases.

level iso-surface. The error tolerance is measured in pixels on the view screen. The lower resolution dataset is a set of tets, possibly from different levels of the hierarchy that approximates the volume dataset to within the error tolerance. This set of tets is free from cracks and T-intersections, and defines a C^0 continuous, piecewise linear approximation to the original data. The iso-surface is extracted from the tets in this lower resolution representation using linear interpolation. The precomputation of min/max ranges over groups of tets called *Diamonds* (Section 2.1) allows portions of the volume that do not contain the iso-surface to be culled quickly.

Crack fixing between levels of detail is done easily within the mesh subdivision scheme. Space subdivision schemes such as octrees require separate steps to fix the cracks in the surfaces that result when cells from different levels in the octree share a face. The tet mesh subdivision scheme easily solves this problem by ensuring that a face touches at most two tets at all times. The crack fixing algorithm is described in Section 2.4.

In Sections 4 and 5, we describe the data structures used to implement the split/merge refinement, and give an efficient, compact method to encode the mesh's structure. This encoding of the mesh structure allows the refinement process to be implemented with fast integer operations.

Fast data location and retrieval are achieved by reordering the data on disk and in main memory to follow the ordering of data indicated by the structure of the mesh subdivision. The data layout scheme is discussed in Section 6. Since the view-dependent refinement is driven by the way the volume dataset is subdivided, ordering the dataset on disk and in memory to match this order improves the performance of the memory system which is important when dealing with large datasets.

2 Longest Edge Bisection Subdivision

Here we establish terminology to describe the mesh components. A tet is described by a *level* and a *phase* based on when it is created in the mesh refinement. The process begins at level 0, phase 0 with the initial configuration of a cube divided into 6 tets. Figure 2 illustrates the three phases of the refinement process. After three subdivisions, the level is incremented by 1. After n subdivisions, the phase is n mod 3 and the level is $\lfloor n/3 \rfloor$. The number of subdivisions is called the *subdivision level*. In the following sections, level refers to the $\lfloor n/3 \rfloor$ not the *subdivision level*. The *split edge* of a tet is the tet's longest edge. In each phase, a tet is subdivided into two *children* at the midpoint of the split edge. This midpoint is called the tet's *split vertex*. A tet can be referenced either by its split edge or its split vertex and the other two vertices not on the split edge.

2.1 Diamonds

When a tet is split, all the tets around its split edge need to be split to fix the continuity on the mesh and avoid T-intersections. A group

Type	Tets(phase,level)	Parents	Children
0	6(0,L)	3(2,L-1)	6(1,L)
1	4(1,L)	2(0,L)	4(2,L)
2	8(2,L)	4(1,L)	8(0,L+1)

Table 1: Number,phase, and level of tets, parents, and children for each type of diamond. L is the *level* of the diamond.

of tets that share a split edge is called a *diamond*. A diamond's *split edge* and *split vertex* are equivalent to the split edge and split vertex of its tets. All diamonds in the mesh can be uniquely identified by their split edge or split vertex. In later sections, a diamond will be referenced by its split edge or split vertex.

Tets are grouped into diamonds to simplify the refinement process so that cracks in the mesh are easily fixed. Simple recursive top-down refinement schemes that subdivide tets can introduce cracks or T-intersection to the mesh unless they take care to refine all of the tets that share a split edge at the same time. By grouping tets into diamonds, we can easily locate all tets around a split edge. Splitting a diamond is equivalent to splitting all of the tets in the diamond. The crack free refinement process is described in detail in Section 2.4 All tets within a diamond have the same level and phase. The *phase* and *level* of a diamond are equivalent to the phase and level of a tet. There is one type of diamond for each phase. Table 1 lists the number of tets, their phase and level for each diamond type.

1. The Phase 0 diamond is shown in Figure 3. The split edge is $(sv0, sv1)$. There are six tets around a major diagonal of a cube. One of the tets is $sv0, sv1, v2, p1)$, the other 5 can be created by rotating this tet around the axis $sv0, sv1$.

2. The Phase 1 diamond is shown in Figure 4. There are four phase 1 tets around a face diagonal. One of the tets is $(v0, v1, v2, v3)$, the other three tets can be created by rotating tet this tet 90, 180, and 270 degrees around the diamond's split edge $(v0, v1)$.

3. The Phase 2 diamond is shown in Figure 5. There are eight tets around an edge of a cube. The eight tets in the diamond can be created by rotating tet $(v0, v1, v2, v3)$ around the diamond's split edge $(v0, v1)$ in 45 degree increments.

There is one diamond associated with each data point in the volume dataset. The 8 diamonds at the corners of the root diamond (i.e. the diamond at level 0 phase 0) are allocated but never used. The diamond contains the following information:

1. The type of the diamond based on the direction of its split edge.

2. Whether or not the diamond is a leaf, on a boundary of the root diamond, or a root diamond.

The type of a diamond is determined by its split edge (SV_0, SV_1), where SV_0 is the first vertex on the split edge. There are 26 different directions; there are 8 different directions for the phase 0 diamonds, 12 for the phase 1 diamonds (4 each on the XY, XZ, and YZ planes), and 6 for the phase 2 diamonds. For example, the split edge from (64,64,0) to (64,0,64) gives the vector (0,-64,64). This corresponds to the direction vector (0,-1,1). The 26 directions come from the different combinations for a vector (i, j, k) when the entries are restricted to -1, 0, and 1. The vector $(0,0,0)$ is not a valid split edge because it indicates that the vertices on the split edge are identical.

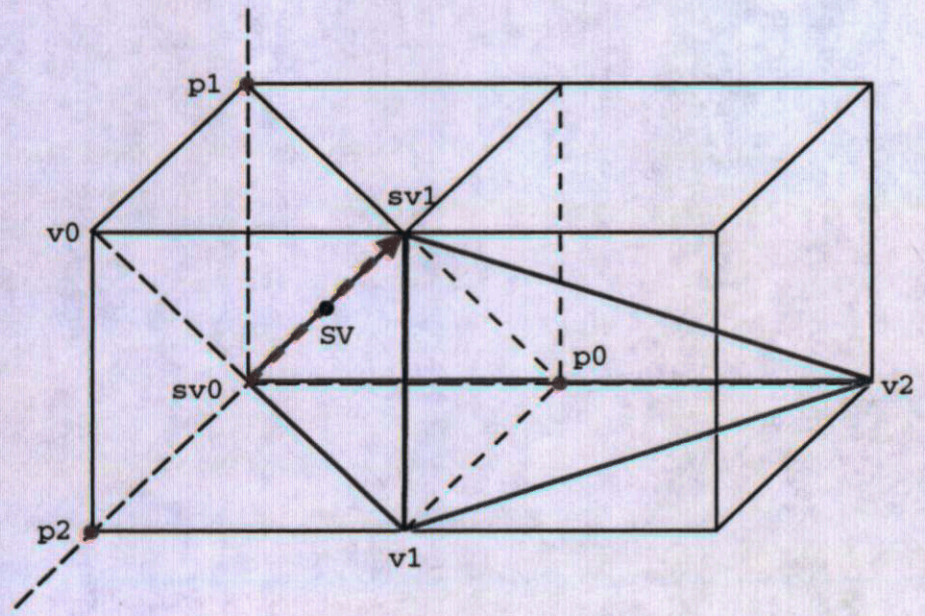

Figure 3: Phase 0 diamond and its parents.

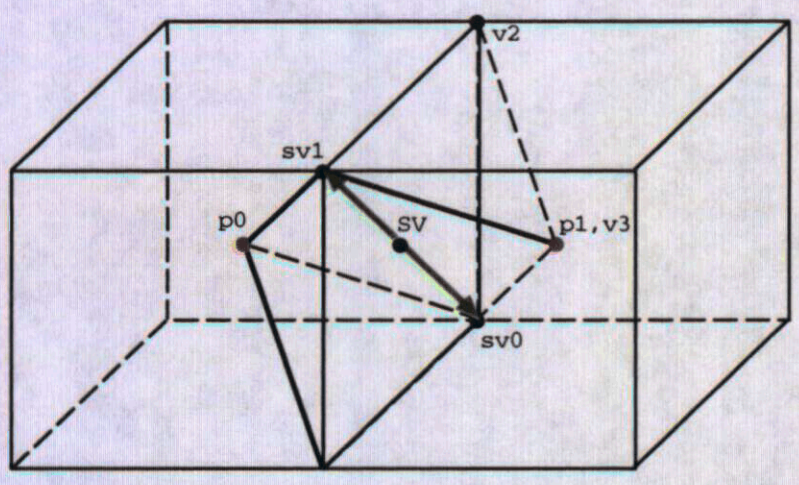

Figure 4: Phase 1 diamond and its parents.

2.2 Parent Diamonds

A tet at subdivision level i is created when a tet from subdivision level $i-1$ is subdivided. For a diamond D, the diamonds that need to be split to create all of D's tets are called D's *parents*. The parents of a diamond are diamonds from the previous subdivision level. The parent information is summarized in Table 1. Figures 3 - 5 show the parents for the three types of diamonds. In all of the pictures, the split edge is $(sv0, sv1)$, the split vertex is SV, and the parents are shown as $p0, p1, \ldots$. As stated earlier, a diamond is referenced by its split vertex.

2.3 Child Diamonds

For a diamond D, the diamonds that are created when D is split are called D's *children*. The children of a diamond are diamonds from the next subdivision level. The child information is summarized in Table 1.

1. *Phase 0:* The children are located at the centers of the diamond's exterior faces. Three of the children of a phase 0 diamond are shown in Figure 6. The other three exist on the other faces of the cube.

2. *Phase 1:* The children are located at the centers of the edges of the face containing the split vertex. The children for phase 1 diamonds in the XZ and YZ plane are shown in Figure 7. Diamond p0 has children c0 - c3, and diamond p1 has children c3 - c6. The children for diamonds in the XY plane are found in a similar manner.

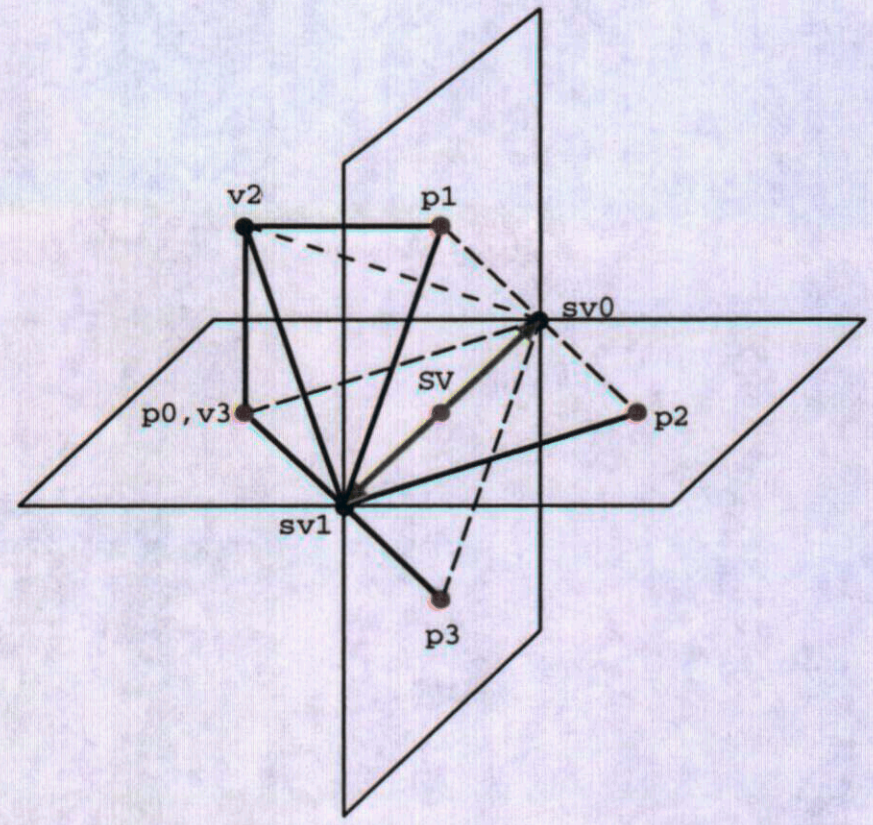

Figure 5: Phase 2 diamond and its parents.

3. *Phase 2:* The children are the diamonds from level $i+1$ that touch the parent diamond's split edge. In Figure 8, four of the children for diamond p0 are c0 - c3. The other four children are the centers of the four octants that come out of the page. The red arrows indicate the split edges of the children.

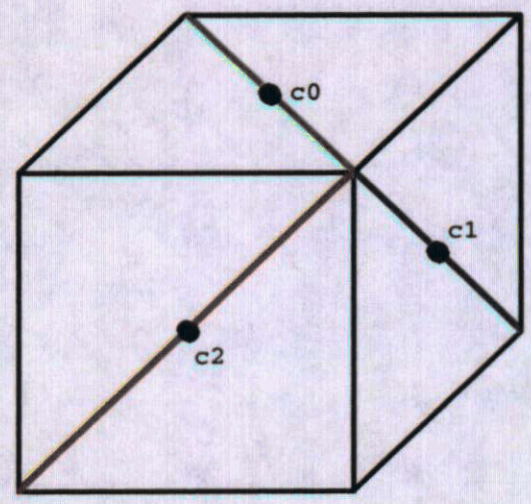

Figure 6: Children of Phase 0 diamond.

2.4 Split/Merge Refinement

As described in Section 1, the tet mesh supports the dual queue split/merge refinement strategy as described in the ROAM system[Duchaineau et al. 1997]. In our system, we use this strategy because it provides more frame to frame coherence than the top down split only algorithm. In most interactive applications, the viewing position does not change a significant amount between consecutive frames. In frame $i+1$, many diamonds from frame i will have a view-dependent error that is still within the current error tolerance. These diamonds can be reused in frame $i+1$. A small fraction of the diamonds will need to be split or merged to satisfy the error tolerance. In most cases, a top down algorithm that ignores the mesh from frame i would perform a much larger number of splits to create the mesh for frame $i+1$ than the number of splits and merges

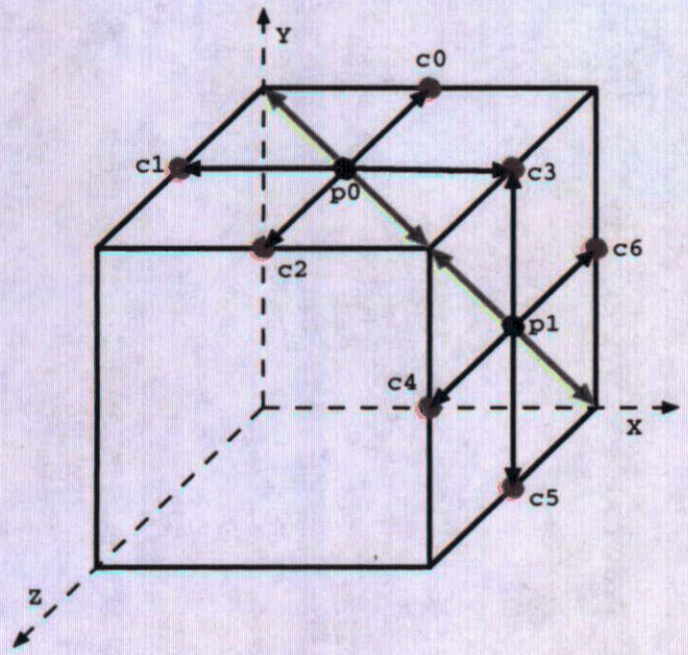

Figure 7: Children of Phase 1 diamond.

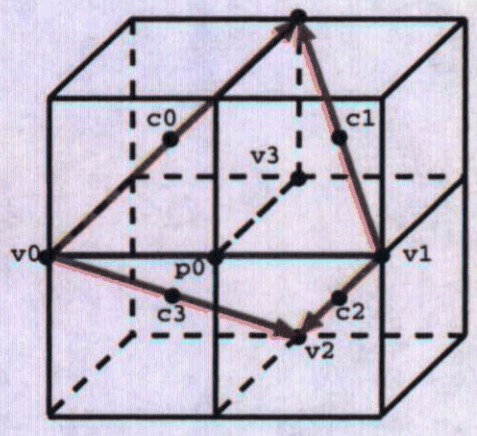

Figure 8: Children of Phase 2 diamond.

performed by the split/merge algorithm. By starting the refinement process for frame $i+1$ with the mesh from frame i instead of the base mesh, a large number of splits and merges do not have to be done.

In a preprocessing stage, we compute the following information, necessary to drive the refinement process, for each diamond in the hierarchy:

1. The diamond information. (See Section 2.1)
2. The iso-surface approximation error. (See Section 3)
3. The min and max data values within the diamond including the diamond boundary. These are used to quickly determine if a diamond contains the iso-surface during mesh refinement.
4. The gradient vector at the split vertex. The gradient vectors are normalized and quantized into a single word.

The computation of the diamond information, min/max values and gradients runs in $O(ND)$ time, where ND is the number of diamonds in the mesh. The diamond information and min/max values are computed by traversing the mesh in a top-down fashion. The computation of the error values runs in $O(NT)$ time, where NT is the number of tets in the mesh. The number of tets is 6 times the number of diamonds so the computation of the error values becomes fairly expensive for large datasets. The computations can be easily parallelized, and were done on a multiprocessor SGI machine.

The *current mesh* is a set of tets that approximates the volume dataset to within a certain view-dependent error bound. The split

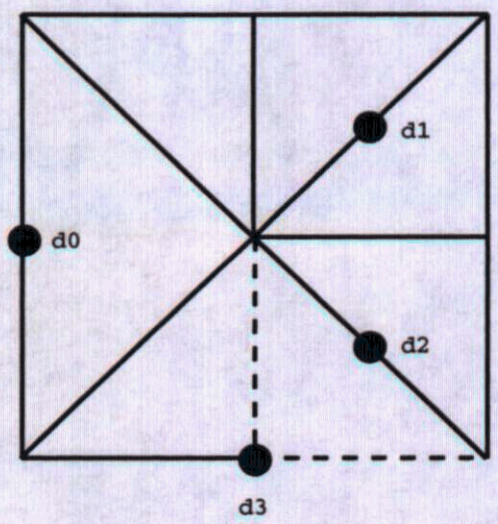

Figure 9: Diamond $d1$ has 2 triangles in the mesh. Diamond $d2$ has one triangle in the mesh.

queue holds the diamonds that these tets belong to. Each diamond in the split queue contains flags indicating which of its tets are actually in the *current mesh*. These flags are called the diamond's *tet flags*. The reason for these flags is illustrated for the 2D case in Figure 9. The mesh has four diamonds $d0-d3$. Diamond $d1$ has two triangles that are both in the mesh. Diamond $d2$ has two triangles only one of which is in the current mesh. The triangle not in the mesh is shown with the dashed lines. The diamond's *tet flags* are used to record this information. After the refinement process, the iso-surface is extracted from the tets in the *current mesh*. This is done by going through the diamonds split queue and examining its *tet flags* to find the tets in the *current mesh*. The merge queue holds the diamonds that can be merged. These are diamonds that have been split and whose children have not been split.

At frame 0 the split queue is initialized with the base configuration of 6 tets (i.e. the root diamond). The merge queue is empty. At each frame, given a view-dependent error tolerance E, the following steps are done when the viewpoint changes:

1. Diamonds not within the view frustum are marked as *invisible* and diamonds that do not contain the iso-surface are marked as *empty*; they are assigned a view-dependent error of 0. View-dependent errors are recomputed for all other diamonds in the split and merge queues.
2. Diamonds in the split queue whose error is greater than E are split. Diamonds in the merge queue whose error is less than E are merged. *Invisible* and *empty* diamonds in the split queue are never split. In the merge queue, they are the first diamonds to be merged.
3. The refinement process is stopped when all diamonds in the split queue have an error below E and all diamonds in the merge queue have an error above E, or when the time allowed for processing the current frame has elapsed.
4. The iso-surface is extracted from the tets in the *current mesh* that belong to the visible, non-empty diamonds in the split queue.

A diamond is split by splitting all of its tets and adding them to the split queue. A tet is added to the split queue by creating an entry in the split queue for its diamond and then setting the flag which indicates that the tet is in the current mesh. When some of the tets in a diamond do not exist (i.e. they are not in the current mesh), it is necessary to create them before they can be split. This is done by splitting the diamond's parents that have not been split. Lastly the diamond is removed from split queue and added to the merge queue.

A diamond is merged by merging all of its tets and then adding them to the split queue. A tet is merged by removing its two children from the split queue. A tet is removed from the split queue by locating its diamond's entry in the split queue and unsetting the flag that was set when the tet was added to the split queue. When a tet is removed from the mesh, its diamond is checked to see if it has any more tets in the current mesh. If not, the diamond is removed from the split queue. Lastly, the diamond's parents are checked to see if they can be added to the merge queue. A diamond can be added to the merge queue only if all of its children are in the split queue.

3 Error Metrics

Each diamond in the mesh has an approximation error, iso-surface error, and a view-dependent error associated with it. The approximation error ae_T for a tet is the maximum difference between the linear approximation over the tet and the actual data values for the points inside the tet. The approximation error ae_D for a diamond is the maximum of its tets' approximation errors. Leaf tets have an approximation error of 0 as do leaf diamonds.

The iso-surface error of a tet is the maximum deviation of an iso-surface drawn through the tet from the actual iso-surface passing through the tet. This calculation is illustrated in Figure 10 for the 1D case. The original function is $f(x)$ and it is approximated by $l(x)$. The upper and lower bounds on the approximation, given by the approximation error ae, are $a1(x)$ and $a2(x)$. The iso-contour for a given function value y is drawn. The iso-contour using $l(x)$ occurs at point a where $y = l(a)$. The actual iso-contour using $f(x)$ occurs the point b where $y = f(b)$. The error in the iso-contour is given by:

$$ie = |a - b| \tag{1}$$

An upper bound u for the iso-surface error can be computed by:

$$u = ae/m, \tag{2}$$

Where ae is the approximation error and m is slope of the linear approximation l. As the slope of l increases, f converges to a vertical line and is approximated with increasing accuracy by l. This means that the approximation errors get smaller and smaller. As the slope of l decreases, the iso-contour approximation a and the actual iso-contour b for a function value y can be far apart even if ae is small. In higher dimensions, the slope of the approximation translates to the magnitude of the gradient. In 3D this is the gradient of the field through a tetrahedron as given by its linear approximation. The iso-surface error is clamped at the physical size of the tet because the iso-surface drawn through a tet can never be outside the tet's boundaries. The iso-surface error for a tetrahedron T is given by:

$$ie_T = ae_T / \|\nabla T\|, \tag{3}$$

The iso-surface error ie_D for a diamond is the maximum of its tets' iso-surface errors. The view-dependent error of a diamond is the projection of its iso-surface error onto the view screen. This is done by creating a sphere at the diamond's split vertex of radius ie_D and projecting this sphere onto the view screen. The size of the projected sphere (i.e. width or height in pixels) is the view-dependent error. Details on view-dependent error metrics can be found in [Hoppe 1997] , [Lindstrom and Pascucci 2001], and [Luebke and Erikson 1997]. As described in Section 2.4 the view-dependent error is updated for all diamonds within the view-frustum at the start of each frame.

4 Data Structures

As stated in Section 2.4, we precompute the diamond information, iso-surface approximation error, min, max values, and gradient vector for each diamond. These values are stored on disk and mapped to main memory at run time using the Unix **mmap** command. The mmap function establishes a mapping between a process's address space and a virtual memory object represent as a disk file. This allows us to keep in memory only the data that is currently being used by the split/merge process and the iso-surface extraction process. The diamond information is stored in one file. The data values and gradients are stored in another file because they are used for extracting and drawing the iso-surface but not for mesh refinement. The error values, min, and max values are stored in another file because they are used together in the mesh refinement process but not for iso-surface extraction.

The split and merge queues are implemented as hash tables using a fixed number of buckets and chaining to handle collisions. Each bucket corresponds to a range of the projected screen space error as measured in pixels. Each entry in the bucket corresponds to a diamond whose screen space error falls within the bucket's range. At run time, a small fraction of the diamonds in the hierarchy are in the split and merge queues. A separate hash table, the queue hash table, is used to map diamond indices to their entries in the queue. There is one hash table for the split queue and one hash table for the merge queue. This second hash table is necessary because the split and merge queues are ordered by view-dependent error. In or-

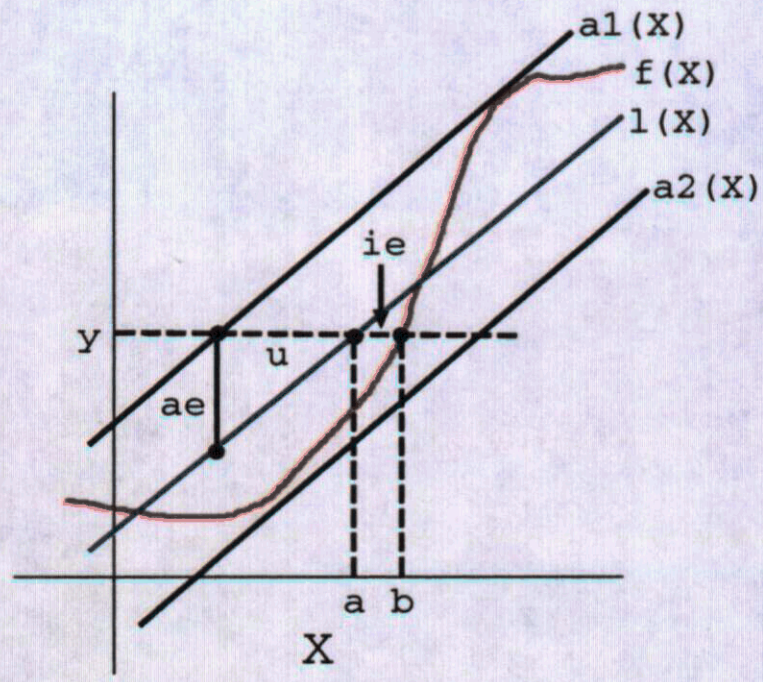

Figure 10: Iso-Surface error calculation in 1D.

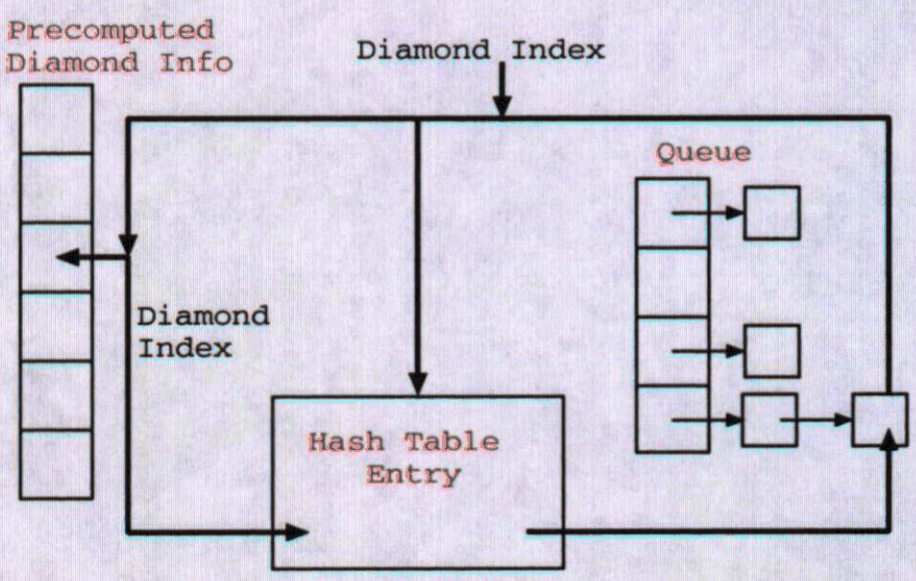

Figure 11: Relationship between precomputed data, queue entries, and queue hash table.

der to quickly locate a specific diamond in either queue, we need to be able to access the queue based upon the (i,j,k) index of the diamond which uniquely identifies a diamond. Accessing the diamonds in the queues based on view-dependent error would require computing the view-dependent error, locating the bucket that the diamond is in, and then traversing the bucket to get the appropriate entry.

The data structures are illustrated in Figure 11. The hash table maps a diamond index to an entry in the queue. The diamond index associated with the queue entry maps back to the precomputed per diamond information and the same hash table entry. When a tet is added or removed from the mesh, its diamond's index is used to locate the corresponding entry in the split queue via the split queue's hash table. The diamond's *tet flag* associated with this tet is set or unset. The flags are implemented as a single bit, and correspond to the precomputed cell encodings for the diamond as described in Section 5. Each bucket entry in the split and merge queues stores the diamond's level, (i,j,k) index, iso-surface and view-dependent errors, *invisible* and *empty* bits (Section 2.4), and diamond information (Section 2.1), and its *tet flags*. The iso-surface error and diamond information are copied from the mmaped arrays of precomputed data.

Interpolation caching, computing the location of the iso-surface within a tet and saving the interpolated positions and normals, is done on a per tet basis as opposed to a per edge basis as done in [Gerstner and Rumpf 2000]. When a tet is added to the split queue, the iso-surface through it is computed and stored in the *triangle cache*. Each element in the triangle cache stores 4 vertices and 4 normals vectors. The vertices are stored as shorts and the normal vectors are stored as 10 bit quantities in a single word. The triangles are cached in an array so that they are all in a contiguous region of memory. New triangles are appended to the end of the array. Triangles are removed from cache by replacing the removed triangle with the triangle at the end of the array. This caching method does not reuse interpolations along edges, and it duplicates normals and vertices along edges. Its advantage is that it has better memory coherence than hash table based caches. In each frame the mesh is drawn simply by traversing the *triangle cache*.

5 Mesh Encoding

The mesh structure can be encoded in a very compact manner assuming that the data points lie on a $2^n+1 \times 2^n+1 \times 2^n+1$ grid. This is because the offsets to compute the tet vertices, parents and children of a diamond are all powers of two relative to the split vertex of the diamond. Data that do not lie on this type of grid can be embedded in a *virtual grid* of the proper size.

A diamond is represented as an (i,j,k) index. This index is how the diamond information would be accessed if it were stored in a C-style 3D array. The vertices on the split edge of a diamond are encoded in a single byte as an offset vector from the split vertex. For example, the split edge with $SV_0 = (64,64,0)$ and $SV_1 = (64,0,64)$ has the vector $(0,-64,64)$ and split vertex $(64,32,32)$. Normalizing this vector yields $(0,-1,1)$. These values are stored as 2 bit quantities in a single byte. SV_0 and SV_1 are computed by rescaling the normalized vector and adding/subtracting it from the split vertex. In this case,$(0,-1,1)$ would be rescaled to $(0,-32,32)$. The rescaling factor is easily determined from the level of the diamond. For a mesh with l levels, the scaling factor for a diamond at level j is given by: $2^{(l-j-1)}$. (The subdivision starts at level 0.) Since this factor is always a power of 2, computing the indices is done using only shifts and adds. The split edge encodings are stored in a lookup table and accessed at run time based upon the type of the diamond. Since a diamond is identified by its split vertex, the vertices on the split edge can be computed by knowing the diamond's type and level.

Time	Operation	# Elements	Elem/Sec
0.048	Cull/Priority	47,339	976,606
0.140	Colored Draw	50,001	357,692
0.173	Textured Draw	50,001	289,023
0.01	Split/Merge	15-150	1500-15,000

Table 2: Timings results for algorithm sections.

The parents, tets, and children of a diamond are also encoded relative to the split vertex of the diamond and stored in a lookup table. Since two of a tets's vertices are the vertices on the split edge, they do not need to be encoded again. The other two vertices of the tet are encoded and stored in a lookup table in the same way that the split edge is encoded. For any diamond, the (i,j,k) index for a parent, child or tet vertex can be computed from the diamond's split vertex and the proper encoding. There is one set of encodings for each of the 26 types of diamonds. Using these indices and offsets to access the diamonds eliminates the need for a diamond to have pointers to its parents and children.

6 Memory Layout

In order to improve cache performance and effectively utilize the available memory bandwidth we arrange our data on disk and in memory in a manner that follows the data ordering indicated by the mesh subdivision. This data layout scheme and its performance benefits are detailed in [Pascucci 2000]. [Lindstrom and Pascucci 2001] describe different layout schemes for terrain rendering applications that are easily extended to volumetric data. Out scheme is based on hierarchical Lebesgue Z space filling curves and is called $z-order$ because the curve looks like the letter Z. The (i,j,k) indices used in the mesh refinement process are easily converted to the indices for the new data layout scheme via lookup tables. The precomputed data described in Section 2.4 is stored on disk in this manner.

7 Results

We have tested our algorithms on an SGI Octane workstation with two 300 MHZ R12000 processors, 896 MB of memory and an ESI graphics board and an SGI Onyx with 44 250 MHZ R10000 processors and IR2 graphics boards. At run time the algorithm uses one processor and one graphics pipe. The preprocessing was done in parallel on an SGI Onyx.

Table 2 shows timings results for the individual sections of the algorithm. The Cull/Priority operation is the recomputation of the diamond's visibility and view-dependent errors. For these results there were no culled diamonds; the visibility test and the error calculation were done for every diamond in the split and merge queues. The colored draw and textured draw operations render the geometry using colors and textures respectively. The objects are textured using a sphere map. The Split/Merge operation is the number of splits and merges that can be done in a period of time. For these results we fixed the time frame at 10ms. The large variance in the number of splits and merges is due to the varying complexity of the operations. Some splits may have to locate and split many parents, and some merges may have to locate and examine many children. These results are taken from the SGI Octane workstation.

The first example is the rocker arm dataset taken from the Cyberware samples page. The initial polygon model was converted to a 256x256x256 distance field with floating point samples. The full resolution iso-contour contains 1,586,604 triangles. Our second example is the igea head dataset also from the Cyberware samples

Dataset	Machine	Format	Dims	Tris/Frame
Rocker Arm	Octane	float	256^3	250K
Igea Head	Onyx IR2	float	512^3	500K

Table 3: Triangle counts for test datasets.

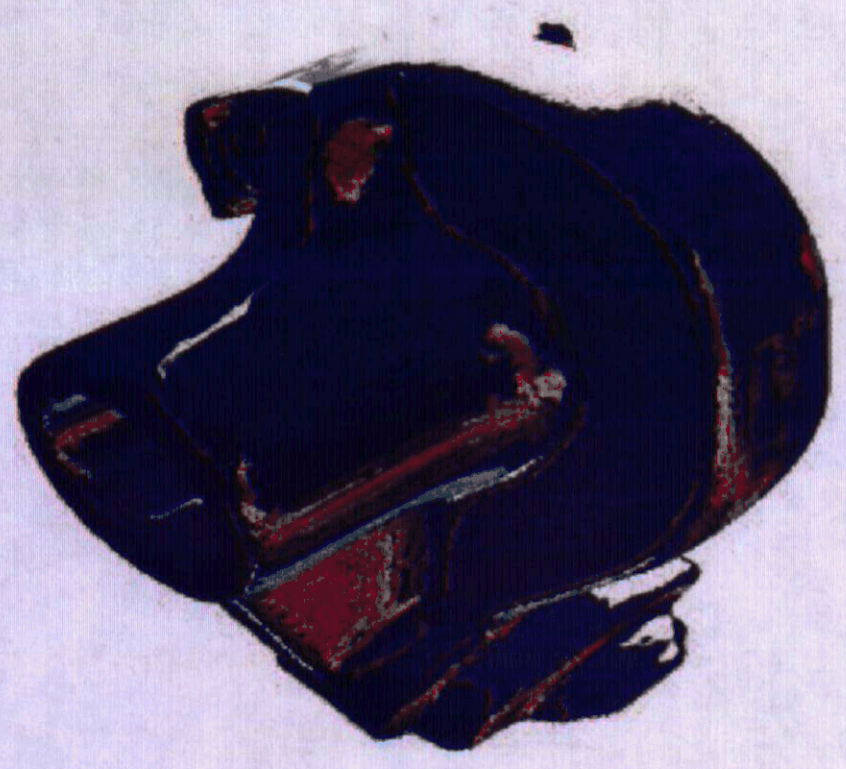

Figure 12: Original Iso-surface for 256^3 rocker arm dataset.(1,350,292 triangles)

page. The polygon model was converted to a 512x512x512 distance field using floating point data values. Table 3 shows triangles per second results for the two datasets. The superior graphics performance of the Onyx makes the drawing time equal to the culling and priority recomputation time. On the Onyx drawing 50K triangles takes about 0.05s. On the Octane the drawing time is about 4 times as slow. When combined with the time for computing visibility and priorities, this gives about a 2x increase in triangles per second.

Figure 12 shows the full resolution iso-surface of the rocker arm dataset. Figures 13 and 14 show the triangulations of the dataset rendered with pixel errors of 1.3 and 0.8 pixels respectively. Figure 1 show the igea dataset rendered at a screen error 0.56 pixels. Environment mapping is used to texture the model.

Figure 12 shows the full resolution iso-surface of the rocker arm dataset. Figures 13 and 14 show the triangulations of the dataset rendered with pixel errors of 1.3 and 0.8 pixels.

8 Conclusions and Future Work

We have presented an algorithm for quickly extracting and rendering iso-surfaces from large volume datasets. Our algorithm extends a multi-resolution tetrahedral mesh based on edge bisection to support adaptive refinement and coarsening. We have shown an efficient way to encode the tets, parents, and children of the mesh structure so that the mesh can be represented compactly and with few pointers. The implementation of the dual queue split/merge algorithm utilizes this new encoding of the mesh through the addition of a queue hash table which enables the queues to be accessed by view-dependent error and diamond indices. Our algorithm easily integrates with other optimizations such as view-frustum culling, deffered priority recomputation, and interpolation caching.

Areas for future work include: topology preserving and simplification algorithms such as those presented in [Gerstner and Pa-

Figure 13: Iso-surface for 256^3 rocker arm. (Screen Error = 1.3 pixels)

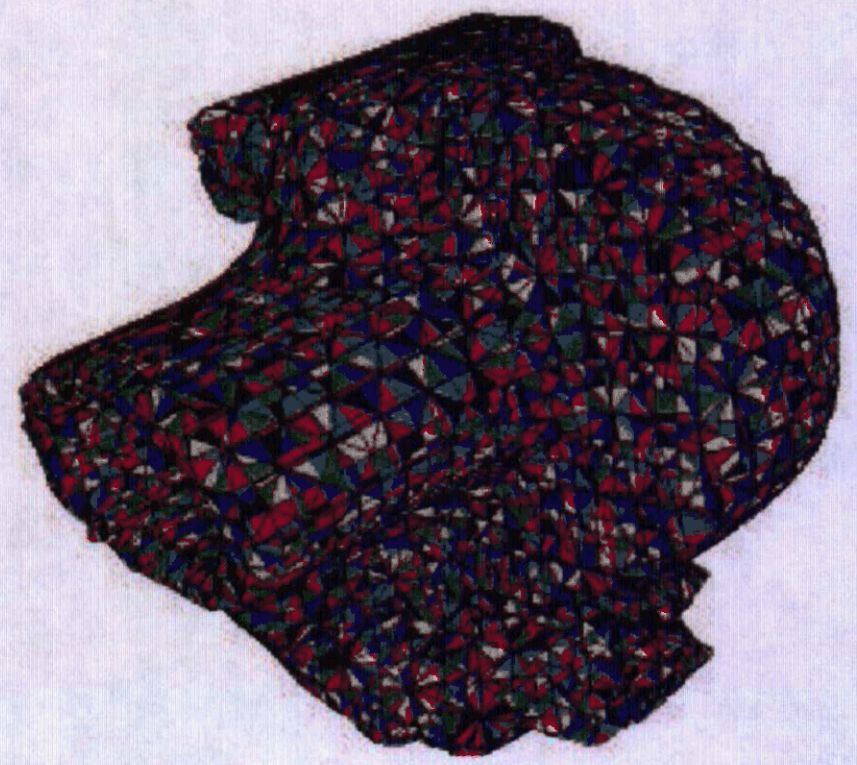

Figure 14: Iso-surface for 256^3 rocker arm. (Screen Error = 0.8 pixels)

jarola 2000] into the refinement process; rendering multiple isosurfaces with transparency as well using volume rendering techniques for image generation; and adapting the mesh structure and refinement schemes to non power of two grids.

9 Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

BAJAJ, C. L., PASCUCCI, V., AND SCHIKORE, D. R. 1996. Fast isocontouring for improved interactivity. In *1996 Volume Visualization Symposium*, IEEE, 39–46. ISBN 0-89791-741-3.

BAJAJ, C. L., PASCUCCI, V., THOMPSON, D., AND ZHANG, X. Y. 1999. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of the 1999 IEEE Parallel Visualization and Graphics Symposium (PVGS'99)*, ACM Siggraph, N.Y., S. N. Spencer, Ed., 97–104.

CARR, J. C., BEATSON, R. K., CHERRIE, J. B., MITCHELL, T. J., FRIGHT, W. R., MCCALLUM, B. C., AND EVANS, T. R. 2001. Reconstruction and representation of 3D objects with radial basis functions. In *SIGGRAPH 2001, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH, E. Fiume, Ed., Annual Conference Series, ACM, 67–76.

CHIANG, Y., AND SILVA, C. T. 1997. I/O optimal isosurface extraction. In *IEEE Visualization 97*, R. Yagel and H. Hagen, Eds., IEEE, 293–300.

CIGNONI, P., MONTANI, C., PUPPO, E., AND SCOPIGNO, R. 1997. Multiresolution representation and visualization of volume data. 352–369. ISSN 1077-2626.

CIGNONI, P., COSTANZA, D., MONTANI, C., ROCCHINI, C., AND SCOPIGNO, R. 2000. Simpli£cation of tetrahedral meshes with accurate error evaluation. In *IEEE Visualization 2000*, 85–92. ISBN 0-7803-6478-3.

DUCHAINEAU, M. A., WOLINSKY, M., SIGETI, D. E., MILLER, M. C., ALDRICH, C., AND MINEEV-WEINSTEIN, M. B. 1997. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization '97*, IEEE Computer Society Technical Committee on Computer Graphics.

FERLEY, E., CANI, M.-P., AND GASCUEL, J.-D. 2000. Practical volumetric sculpting. In *The Visual Computer*, vol. 16(8). Springer, 469–480.

FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., AND JONES, T. R. 2000. Adaptively sampled distance £elds: A general representation of shape for computer graphics. In *Siggraph 2000, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., Annual Conference Series, ACM, 249–254.

GERSTNER, T., AND PAJAROLA, R. 2000. Topology preserving and controlled topology simplifying multiresolution isosurface extraction. In *Proceedings Visualization 2000*, T. Ertl, B. Hamann, and A. Varshney, Eds., IEEE Computer Society Technical Committee on Computer Graphics, 259–266.

GERSTNER, T., AND RUMPF, M. 2000. Multiresolution parallel isosurface extraction based on tetrahedral bisection. In *Volume Graphics*, Springer, M. Chen, A. Kaufman, and R. Yagel, Eds., Computer Graphics Proceedings, Annual Conference Series, ACM, 267–278.

HOPPE, H. 1997. View-dependent re£nement of progressive meshes. In *SIGGRAPH 97 Conference Proceedings*, Addison Wesley, T. Whitted, Ed., Annual Conference Series, ACM SIGGRAPH, 189–198. ISBN 0-89791-896-7.

KOBBELT, L. P., BOTSCH, M., SCHWANECKE, U., AND SEIDEL, H.-P. 2001. Feature-sensitive surface extraction from volume data. In *SIGGRAPH 2001, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH, E. Fiume, Ed., Annual Conference Series, ACM, 57–66.

LINDSTROM, P., AND PASCUCCI, V. 2001. Visualization of large terrains made easy. In *Proceedings of IEEE Visualization 2001*, T. Ertl, K. Joy, and A. Varshney, Eds., IEEE Computer Society Technical Committee on Computer Graphics.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, vol. 21, 163–169.

LUEBKE, D., AND ERIKSON, C. 1997. View-dependent simpli£cation of arbitrary polygonal environments. In *Proceedings of SIGGRAPH 97*, ACM SIGGRAPH / Addison Wesley, Los Angeles, California, Computer Graphics Proceedings, Annual Conference Series, 199–208. ISBN 0-89791-896-7.

PASCUCCI, V. 2000. Multi-resolution indexing for out-of-core adaptive traversal of regular grids. In *Proceedings of the NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometric Methods for Scienti£c Visualization*, H. Hagen, G. Farin, and B. Hamann, Eds. Available as LLNL technical report UCRL-JC-140581.

TROTTS, I. J., HAMANN, B., AND JOY, K. I. 1999. Simpli£cation of tetrahedral meshes with error bounds. 224–237. ISSN 1077-2626.

ZHOU, Y., CHEN, B., AND KAUFMAN, A. 1997. Multiresolution tetrahedral framework for visualizing regular volume data. In *IEEE Visualization '97*, IEEE Computer Society Technical Committee on Computer Graphics.